An Alternative Method For Performing Board Level Simulations involving Microprocessors

Gary Bolotin

Jet Propulsion laboratory
California Institute of Technology
4800 oak Grove Drive
Pasadena, CA 91109
bolotin@telerobotics.jpl.nasa.gov
818 354-4126 FAX 818 393-5007

Abstract

This paper will present an alternative method for performing board level simulations of designs involving microprocessors. This technique makes use of only a standard "C" compiler and simple simulation library functions to perform accurate board level simulations. This technique was used to simulate. the IBM 1750 based processor that interfaced to the ASICs that are being developed for the CASSINI spacecraft. Conventional processor modeling techniques involve the use of hardware modeling or extensive behavioral models of the microprocessor. These techniques are expensive in both cost and resources required.

Summary

When developing ASICs that interface to microprocessors, it is desire.d to perform a board level simulation that shows that the ASICs under development will function at the system level. This requires a simulation model of the processor. This model can be implemented using one of the following methods:

1). Hardware Modeling
2). Behavioral Model of Processor (software - VHDL)
3). Bus level model of processor.

1 will start by briefly describing the first two methods. The third method, the bus level model of the processor will then be discussed in detail.

The hardware modeling technique can vary from using the real processor, with a discrete logic or FPGA implementation of the ASIC, to using a hardware modeling library, similar to the Mentor I] ML. Both Of these methods USC the actual microprocessor to perform the simulation. Although this technique is quilt accurate, it is very expensive.

When using a software mode.1 of the microprocessor a model is developed which will simulate the microprocessor in every mode of operation, i.e.. instruction fetch, execute, prefetching of instructions, cct. All internal registers and interactions need to be modeled. The software that needs to be developed can be quite extensive. This technique is also quilt expensive.

The two techniques just described, all model the microprocessor down at least to the. register level. Both rely on native code being presented to it. Assuming one is simulating a 8086 based system on a simulator running on a SUN workstation, one would need a 8086 cross compiler that runs on the SUN workstation in order to perform the board simulation. The 8086 test software would be compile.d and linked on the SUN. The resulting executable image would then be loaded into the simulator, perhaps by storing it in a ROM model. This is a very time consuming process. In addition the processor will be repeatedly fetching code from memory. An interaction which when shown to work once in the simulation, need not be repeated. Another drawback is that once the processor fetches data of an unknown value, for what ever the reason, the entire simulation is likely to be corrupted.

The technique that is to be described, makes use of the fact that when testing out whether an ASIC will work in the system, the designer is concerned mainly with whether the. ASIC under development will interact correctly with the microprocessor and other dc.vices, not in developing code or a complicated processor model.

The, basic development flow is shown in Figure 1. Test vectors arc produced using a "C" language test file which when run on the simulators host computer, will produce test vectors that can bc applied to the simulator, Using simulator commands these vectors manipulate the pins of the processor in such a way as to simulate its operation in the system. Using a simulator system call, the vectors can also check for proper operation. For example on a read cycle the test vectors can test that proper data has been read back. An error message can be printed if the data read is not what is expected .

As an example, suppose we want to simulate a processor with a very protocol bus scheme as illustrated in figure 2 and 3, and described as follows.

Write Cycle

1). Address and Data arc put on to the bus.
2). The signal ADVn is asserted.
3). Wait for slave. to assert DTACKn.
4). Deassert ADVn.
5). Remove. Address and Data.
6). Wait for DTACKn to be deasserted.

Read Cycle

1). Address is put on to the bus.
2). The signal ADVn is asserted.
3). Wait for slave. to assert DTACKn.
4). Sample Data lines.
5). Deassert ADVn.
6). Remove Address and Data.
7). Wait for DTACKn to be deasserted.

Two basic routines need to be written, one to simulate a read cycle and one to simulate the write cycle. The read routine is called with the address of the data byte 10 be read, and the value that is to be expected. When executed, if the data read is different then what is expected, an error message will be printed in the simulator list window. This is handled by the routine check. output. The routine, check..output is a simulator specific macro that compares a signal to a expected value, and will print an error message if values compared arc not equal. The write routine is called with the address of the data byte to be written and the data value.

```
write(address, data- expected)
unsigned   int address,data_ expected;
{
        printf(''forc.c ADDRESS %x\n,address);
        print f("run   %d\n",TSET)
        printf(''forcc ADVn O);
        printf("force WRn O);
        printf(''brcak DTACKn O);
        printf(''chcck- output(DATA,data_expected)\n");
        printf(''forcc WRn 1);
        printf(''forcc ADVn 1);
        print f("run %d\n",THOLD)
        print f(''forgct  force ADDRESS \n);
        printf("forgct  force DATA \n);
)

read(address, data)
unsigned   int address, data;
{
        prinif(''forcc ADDRESS %x\n,address);
        printf(''forcc  DATA %x\n,address);
        print f("run   %d\n",TSET)
        printf(''forcc RDn O);
        printf(''forcc ADVn O);
        IJ1-illtf(''brcak DTACKn O);
        check. output(DATA,data_expected);
        prinlf(''forcc RDn1);
        prinlf(''forcc ADVn 1);
        print f("run %d\n",THOLD)
        printf(''forgct force ADDRESS /n);
)
```

Wc can now usc the above routines to simulate the processor interfacing with the rest of the system. Instruction fetches can be simulated by reading from ROM or RAM memory spaces. ASIC functionality can be tested by writing to ASIC registers and reading back the expected results.

As an example, we can use the routines just described, to test a block of ASIC registers that are both readable and writeable. We can write a simple loop to perform this function as follows.

```
        . . . . .
        for (addr = START_BLOCK; addr <= STOP_BLOCK, addr+ -t)
              write. (addr, J'A"l''J'l:RN);
              read(addr,PATTERN);
        )
        . . . . .
```

This routine is quite simple. The same routine that is developed to test the ASIC in the simulated environment can be also used to test the ASIC in the real system after the ASIC is fabricated. This is made possible because tile. "C" language is portable from system to system. If on the other hand, the test vectors were written in a simulator specific language, the test vectors developed will have little use outside the simulator.

If wc were to do the same thing using a behavioral model of the processor, wc would first have to compile the code. The code would have to be loaded in the simulation some, how. The simulator would then be told to run. Correct operation would most likely be verified by hand, by examining traces of the performed Read cycles, a very time consuming and error prone process.

Conclusion

This technique for simulating microprocessors is quite simple and elegant and inexpensive. This method frees the user from developing code native to the. target processor thus allowing the user to concentrate on the task at hand; verifying that the. ASIC under development will inte ract correctly with the rest of the system.

References

[1] Zainalabcdin Navabi, "Using VHDL for Modeling and Design of Processing Units", Fifth Annual IEEE International _ ASIC Conference and Exhibit, September 21 -?5, 1 992.

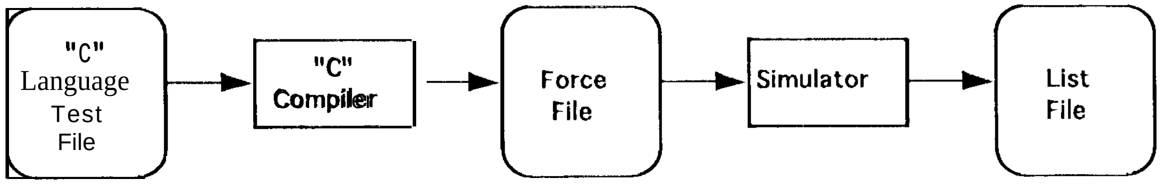

Figure 1. Test Vector Development Flow

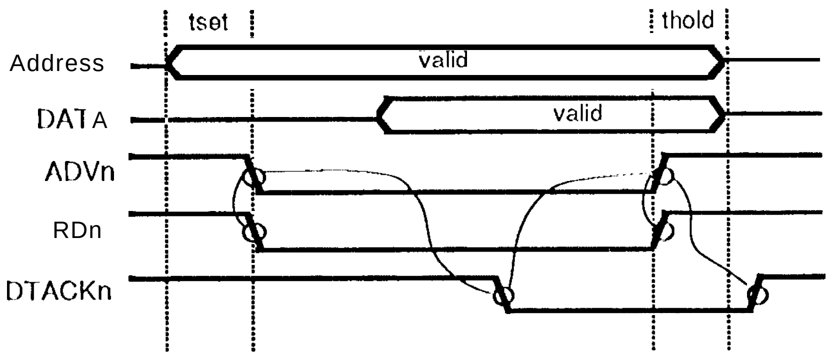

Figure 2. Read Cycle

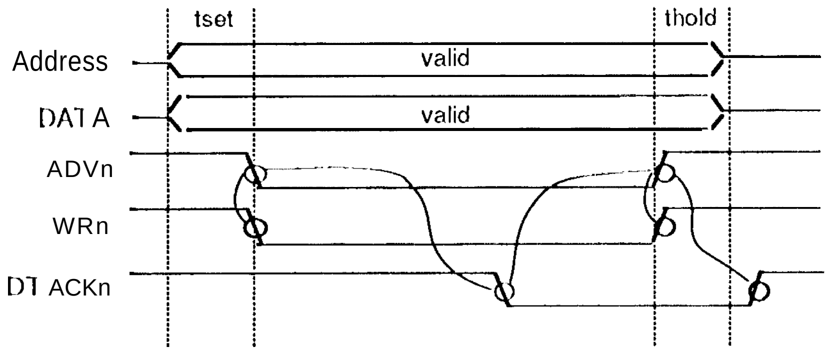

Figure 3. Write Cycle